Analisis de algoritmos y metodos de ordenamiento

Analisis Algoritmico:

El analisis de algoritmos es una parte de la Teoria de complejidad computacional, que provee estimaciones teoricas para los recursos que necesita cualquier algoritmo que resuelva un problema computacional dado. Existen muchos tipos de enfoques al analisar los algoritmos, ya que estos pueden ser fácil de entender, codificar y depurar (Ingeniería de Software) o eficientes con los recursos de la computadora (Análisis y Diseño de algoritmos). El **análisis de algoritmos** nos permite medir la dificultad inherente de un problema y evaluar la eficiencia de un algoritmo.

La caracterıstica básica que debe tener un algoritmo es que sea **correcto**, es decir, que produzca el resultado deseado en **tiempo finito**. Adicionalmente puede interesarnos que sea **claro**, que este **bien estructurado**, que sea **fácil de usar**, que sea **fácil de implementar** y que sea **eficiente.**
Definimos **eficiencia** de un algoritmo como la cantidad de recursos de computo que requiere; es decir, cual es su tiempo de ejecucion y que cantidad de memoria utiliza.
A la cantidad de tiempo que requiere la ejecucion de un cierto algoritmo se le suele llamar **coste en tiempo** mientras que a la cantidad de memoria que requiere se le suele llamar **coste en espacio**. Este concepto de coste nos permitira comparar distintos algoritmos entre ellos para decidir cual es mejor.

Analisis segun tiempo de ejecucion:

El coste en tiempo de un algoritmo depende del tipo de operaciones que realiza y del coste especiıfico de estas operaciones. Una medida que suele ser útil conocer es el tiempo de ejecución de un algoritmo en función de n, lo que denominaremos T(n). Esta función se puede medir físicamente (ejecutando el programa, reloj en mano), o calcularse sobre el código contando instrucciones a ejecutar y multiplicando por el tiempo requerido por cada instrucción.
Prácticamente todos los programas reales incluyen alguna sentencia condicional, haciendo que las sentencias efectivamente ejecutadas dependan de los datos concretos que se le presenten. Esto hace que más que un valor T(n) debamos hablar de un rango de valores:

Tmin(n) <= T(n) <= Tmax(n)

(los extremos son habitualmente conocidos como "caso peor" y "caso mejor", Tmin y Tmax)

Entre ambos se hallará algún "caso promedio" o más frecuente. Cualquier fórmula T(n) incluye referencias al parámetro n y a una serie de constantes "Ti" que dependen de factores externos al algoritmo como pueden ser la calidad del código generado por el compilador y la velocidad de ejecución de instrucciones del computador que lo ejecuta.

Utilizando la definicion anterior caracterizar a la funcion T puede ser complicado. Por ello se definen tres funciones que dependen exclusivamente del tamano de las entradas y describen resumidamente las caracter´ısticas de la funcion T.

Sea An el conjunto de las entradas de tamano n y Tn : An → R la funcion T restringida a An. Los costes en caso mejor, promedio y peor se definen como sigue:

-Coste en caso mejor : Tmin(n) = $min\{ T(a) \mid a \in A \}$

-Coste en caso promedio : T(n) = $\sum_a Pr(a).T(a)$, donde Pr(a) es la probabilidad de ocurrencia de la entrada a.

-Coste en caso peor : Tmax(n) = $max\{T(a) \mid a \in A\}$

Notacion Asintonica:

El interés principal del análisis de algoritmos radica en saber cómo crece el tiempo de ejecución, cuando el tamaño de la entrada crece. Esto es la eficiencia asintótica del algoritmo. Se denomina “asintótica” porque analiza el comportamiento de las funciones en el **límite**, es decir, su tasa de crecimiento.

Orden de complejidad:

Existen 3 maneras de definir la complejidad de un algoritmo, O, Ω y Θ.

A 0 se le llama orden, definiendo que O(f(n)) define un "orden de complejidad" de f(n). Ejemplos:

O(1) orden constante
O(log n) orden logarítmico
O(n) orden lineal
O(n²) orden cuadrático
O(n^i) orden polinomial (i > 2)
O(i^n) orden exponencial (i > 2)
O(n!) orden factorial

Se puede identificar una jerarquía de órdenes de complejidad que coincide con el orden de la tabla anterior; jerarquía en el sentido de que cada orden de complejidad superior tiene a los inferiores como subconjuntos. Si un algoritmo A se puede demostrar de un cierto orden O_i, es cierto que también pertenece a todos los órdenes superiores (la relación de orden cota superior es transitiva).
Antes de realizar un programa conviene elegir un buen algoritmo, donde por bueno entendemos que utilice pocos recursos, siendo usualmente los más importantes el tiempo que lleve ejecutarse y la cantidad de espacio en memoria que requiera.

Es engañoso pensar que todos los algoritmos son "más o menos iguales" y confiar en nuestra habilidad como programadores para convertir un mal algoritmo en un producto eficaz. Es asimismo engañoso confiar en la creciente potencia de las máquinas y el abaratamiento de las mismas como remedio de todos los problemas que puedan aparecer.

Metodos de Ordenamiento:
DEFINICIÓN.
Entrada: Una secuencia de n números , usualmente en la forma de un arreglo de n elementos.

Salida: Una permutación de la secuencia de entrada tal que a'1 ≤ a'2 ≤ … ≤ a'n.

Un método de ordenación se denomina **estable** si el orden relativo de los elementos no se altera por el proceso de ordenamiento. (Importante cuanto se ordena por varias claves).
Los algoritmos de ordenación interna se clasifican de acuerdo con la cantidad de trabajo necesaria para ordenar una secuencia de n elementos: ¿Cuántas comparaciones de elementos y cuántos movimientos de elementos de un lugar a otro son necesarios?

a) Intercambio o metodo de la burbuja mejorado:

   El algoritmo bubblesort, también conocido como ordenamiento burbuja, funciona de la siguiente manera: Se recorre el arreglo intercambiando los elementos adyacentes que estén desordenados. Se recorre el arreglo tantas veces hasta que ya no haya cambios que realizar. Prácticamente lo que hace es tomar el elemento mayor y lo va recorriendo de posición en posición hasta ponerlo en su lugar.

   PseudoCodigo:

```
for (i = n; i >= 1; i--)
        for (j = 2; j <= i; j++)
        {
                if (A[ j-1] > A[ j ])
                        intercambie(A[ j-1 ], A[ j ]);
        }
```

b) Insersion o metodo de la baraja:

   Consta de tomar uno por uno los elementos de un arreglo y recorrerlo hacia su posición con respecto a los anteriormente ordenados. Así empieza con el segundo elemento y lo ordena con respecto al primero. Luego sigue con el tercero y lo coloca en su posición ordenada con respecto a los dos anteriores, así sucesivamente hasta recorrer todas las posiciones del arreglo.

PseudoCodigo:

```
for ( j =2 ; j < n;j++)
        k = A[j]
        i = j – 1
        while ( i > 0 & A[i] > k)
                A[i + 1] = A[i]
                i = i – 1
        A[i + 1] = k
```

c) Seleccion o metodo sencillo:

El método de ordenamiento por selección consiste en encontrar el menor de todos los elementos del arreglo e intercambiarlo con el que está en la primera posición. Luego el segundo más pequeño, y así sucesivamente hasta ordenar todo el arreglo.

PseudoCodigo:

```
for ( j = 1; j < n-1; j++)
        k = j
        for ( i = j + 1; i < n; i++)
                if (A[i] < A[k])
                        k = i
        Intercambiar (A[j],  A[k])
```

d) Rapido o Quicksort:

La idea básica del algoritmo es elegir un elemento llamado pivote, y ejecutar una secuencia de intercambios tal que todos los elementos menores que el pivote queden a la izquierda y todos los mayores a la derecha. Lo único que requiere este proceso es que todos los elementos a la izquierda sean menores que el pivote y que todos los de la derecha sean mayores luego de cada paso, no importando el orden entre ellos, siendo precisamente esta flexibilidad la que hace eficiente al proceso. Hacemos dos búsquedas, una desde la izquierda y otra desde la derecha, comparando el pivote con los elementos que vamos recorriendo, buscando los menores o iguales y los mayores respectivamente.

Basado en el paradigma dividir-y-conquistar, estos son los tres pasos para ordenar un subarreglo A[p … r]:

-Dividir. El arreglo A[p … r] es particionado en dos subarreglos no vacíos A[p … q] y A[q+1 … r]—cada dato de A[p … q] es menor que cada dato de A[q+1 … r]; el índice q se calcula como parte de este proceso de partición.

-Conquistar. Los subarreglos A[p … q] y A[q+1 … r] son ordenados mediante sendas llamadas recursivas a QUICKSORT.

-Combinar. Ya que los subarreglos son ordenados in situ, no es necesario hacer ningún trabajo extra para combinarlos; todo el arreglo A[p … r] está ahora ordenado.

PseudoCodigo:

Para ordenar un arreglo completo A, la llamada inicial es Quicksort(A, 1, n). con n siendo el tamaño del arreglo.

```
Quicksort( A, p, r)
        if (p < r)
                x = A[p]
                i = p - 1
                j = r + 1
                while ( t )
                        while ( A[ j ] > x )
                                j = j - 1
                        while ( A[ i ] < x )
                                i = i + 1
                        if ( i < j )
                                intercambiar ( A[ i ], A[ j ])
                        else
                                q = j
                                break
                Quicksort(A, p, q)
                Quicksort(A, q+1, r)
```

e) Por mezcla o MergeSort:

Se aplica la técnica divide-y-vencerás, dividiendo la secuencia de datos en dos subsecuencias hasta que las subsecuencias tengan un único elemento, luego se ordenan mezclando dos subsecuencias ordenadas en una secuencia ordenada, en forma sucesiva hasta obtener una secuencia única ya ordenada. Si n = 1 solo hay un elemento por ordenar, sino se hace una ordenación de mezcla de la primera mitad del arreglo con la segunda mitad. Las dos mitades se ordenan de igual forma.
PseudoCodigo:

```
ordenarMezcla(A, int izq, int der)
{
        if ( izq < der )
        {
                centro = ( izq + der ) % 2;
                ordenarMezcla( A, izq, centro );
                ordenarMezcla( A, centro+1, der);
                intercalar( A, izq, centro, der );
        }
}
intercalar(A, int a, int c, int b )
{
        k = 0; i = a;
        j = c + 1;
        n = b - a;
        while ( i < c + 1 ) && ( j < b + 1 )
        {
                if ( A[i] < A[j] )
                {
                        B[k] = A[i];
                        i = i + 1;
                }
                else
                {
                        B[k] = A[j];
                        j = j + 1;
                }
                k = k + 1;
        }
        while ( i < c + 1 )
        {
                B[k] = A[i];
                i++; k++;
        }
        while ( j < b + 1 )
        {
                B[k] = A[j];
                j++; k++;
        }
        i = a;
        for ( k = 0; k < n; i++ )
        {
                A[i] = B[k];
                i++;
         }
}
```

Diferencias entre estos:

En el ordenamiento por intercambio, tanto en el caso medio como en el peor de los casos utiliza aproximadamente n2/2 comparaciones y n2/2 intercambios, por lo cual T(n) = O(n2).
En el de insercion, el numero de comparaciones es n(n-1)/2 lo que implica un T(n) = O(n2). La ordenación por inserción utiliza aproximadamente n2/4 comparaciones y n2/8 intercambios en el caso medio y dos veces más en el peor de los casos. La ordenación por inserción es lineal para los archivos casi ordenados.
El de seleccion es similar al de intercambio, utiliza aproximadamente n2/2 comparaciones y n intercambios, por lo cual T(n) = O(n2).
En quicksort, depende de si la partición es o no balanceada, lo que a su vez depende de cómo se elige los pivotes:

-Si la partición es balanceada, QUICKSORT corre asintóticamente tan rápido como ordenación por mezcla.
-Si la partición es desbalanceada, QUICKSORT corre asintóticamente tan lento como la ordenación por inserción.

El peor caso ocurre cuando PARTITION produce una región con n–1 elementos y otra con sólo un elemento. Si este desbalance se presenta en cada paso del algoritmo, entonces, como la partición toma tiempo Θ (n) y σ(1) = Θ (1), tenemos la recurrencia:

$$\sigma(n) = \sigma(n-1) + \Theta(n) = \sum_{k=1}^{n} \Theta(k) = \Theta(n^2).$$

Por ejemplo, este tiempo Θ(n2) de ejecución ocurre cuando el arreglo está completamente ordenado, situación en la cual INSERTIONSORT corre en tiempo O(n).

Si la partición produce dos regiones de tamaño n/2, entonces la recurrencia es:

$$\sigma(n) = 2\sigma(n/2) + \Theta(n) = \Theta(n \log n).$$

Tenemos un algoritmo mucho más rápido cuando se da el mejor caso de PARTITION.
El caso promedio de QUICKSORT es mucho más parecido al mejor caso que al peor caso; por ejemplo, si PARTITION siempre produce una división de proporcionalidad 9 a 1:

-Tenemos la recurrencia σ(n) = σ(9n/10) + σ(n/10) + n. o

-En el árbol de recursión, cada nivel tiene costo n (o a lo más n más allá de la profundidad log10n), y la recursión termina en la profundidad log10/9n = Θ(log n)
-El costo total para divisiones 9 a 1 es Θ(nlog n), asintóticamente lo mismo que para divisiones justo en la mitad.

-Cualquier división de proporcionalidad constante, por ejemplo, 99 a 1, produce un árbol de recursión de profundidad Θ(log n), en que el costo de cada nivel es Θ(n), lo que se traduce en que el tiempo de ejecución es Θ(nlog n).

El tiempo de ejecución de QUICKSORT, cuando los niveles producen alternadamente divisiones buenas y malas, es como el tiempo de ejecución para divisiones buenas únicamente: O(n log n), pero con una constante más grande.

En el caso de MergeSort, la relación de recurrencia del algoritmo es T(1) = 1, T(n) = 2 T(n/2) + n, cuya solución es T(n) = n log n => O(n log n)

BIBLIOGRAFIA:

-WIKIPEDIA
-www.cs.upc.edu
-http://colabora.inacap.cl
-www.cs.cmu.edu
-codeworldtechnology.wordpress.com
-www.geeksforgeeks.org